Shared Memory Programming with OpenMP
(An UHeM Training)

Süha Tuna
Informatics Institute, Istanbul Technical University

29 April 2016
Outline - I

• Shared Memory Systems
  • Threaded Programming Model
  • Thread Communication
  • Synchronisation
  • Parallel Loops
  • Reductions

• OpenMP Fundamentals
  • Basic Concepts in OpenMP
  • History of OpenMP
  • Compiling and Running OpenMP Programs
Outline - II

• Parallel Regions
  • Parallel region directive
  • Some useful functions
  • Shared and Private variables
  • Reductions

• Work sharing
  • Parallel for/DO loops
  • Scheduling for loops
  • Single directive
  • Master directive

• Synchronisation
Shared Memory Systems
Shared Memory Systems

• Threaded programming is most often used on shared memory parallel computers.

• A shared memory computer consists of a number of processing units (CPUs) together with some memory.

• Key feature of shared memory systems is single address space across the whole memory system.
  • every CPU can read or write all memory locations in the system
  • one logical memory space
  • all CPUs refer to a memory location using the same address
Conceptual Model
Real Hardware

• Real shared memory hardware is more complicated than this …

  • Memory may be split into multiple smaller units

  • There may be multiple levels of cache memory

    some of these levels may be shared between subsets of processors

  • The interconnect may have a more complex topology

• … but a single space address is still supported

• Hardware complexity can affect the performance of programs, but not their correctness.
Real Hardware Example
Threaded Programming Model

- The programming model for shared memory is based on the notions of threads
  - threads are like processes, except that threads can share memory with each other (as well as having private memory)

- Shared data can be accessed by all threads

- Private data can only be accessed by the owning thread

- Different threads can follow different flows of control through the same program
  - each thread has its own program counter

- Usually run one thread per CPU/core
  - but could be more
  - can have hardware support for multiple threads per core
Threads (cont.)
Thread Communication

• In order to have useful parallel programs, threads must be able to exchange data with each other

• Threads communicate with each via reading and writing shared data
  • thread 1 writes a value to a shared variable A
  • thread 2 can then read the value from A

• Note: there is no notion of messages in this model
Thread Communication

Thread 1

Program
mya = 23
a = mya

Private data
23

Shared data
23

Thread 2

mya = a + 1

24
Synchronisation

• By default, threads execute **asynchronously**

• Each thread proceeds through program instructions independently of other threads

• This means we need to ensure that actions on shared variables occur in the correct order: e.g.
  
  • thread 1 must write variable A before thread 2 reads it
  
  or

  • thread 1 must read variable A before thread 2 writes it

• Note that updates to shared variables (e.g. \( a = a + 1 \)) are not atomic!

• If two threads try to do at the same time, one of the updates may get overwritten.
Synchronisation Example

Thread 1
load a
add a 1
store a

Thread 2
load a
add a 1
store a

Program
CPU
Registers
Memory
Parallel Loops

- Loops are the main source of parallelism in many applications

- If the iterations of a loop are independent (can be done any order) then we can share out the iterations between different threads

- e.g. if we have two threads and the loop

  ```
  for (i=0; i<100; i++) {
    a[i] += b[i];
  }
  ```

  we could do iteration 0-49 on one thread and iterations 50-99 on the other.
Reductions

• A *reduction* produces a single value from associative operations such as addition, multiplication, max, min, and, or.

• For example:

```c
b = 0;
for (i=0; i<n; i++)
    b += a[i];
```

• Allowing only one thread at a time to update $b$ would remove all parallelism

• Instead, each thread can accumulate its own private copy, then these copies are reduced to give final result

• If the number of operations is much larger than the number of threads, most of the operations can proceed in parallel
OpenMP Fundamentals
What is OpenMP

• OpenMP is an API designed for programming shared memory parallel computers

• OpenMP uses the concepts of *threads*

• OpenMP is a set of extensions to C, C++ and Fortran

• The extensions consist of:
  • Compiler directives
  • Runtime library routines
  • Environment variables
Directives and Sentinels

• A directive is a special line of source code with meaning only to certain compilers

• A directive is distinguished by a sentinel at the start of the line

• OpenMP sentinels are:
  - C / C++: `#pragma omp`
  - Fortran: `!$OMP`

• This means that OpenMP directives are ignored if the code is compiled as regular sequential C/C++/Fortran
Parallel Region

• The *parallel region* is the basic parallel construct in OpenMP

• A parallel region defines a section of a program

• Program begins execution on a single thread (the master thread)

• When the first parallel region is encountered, master thread creates a team of threads (fork/join model)

• Every thread executes the statements which are inside the parallel region

• At the end of the parallel region, the master thread waits for the other threads to finish, and continues executing the next statements
Parallel Region

Sequential part

Parallel region

Sequential part

Parallel region

Sequential part

Program Fred

!$omp parallel

...

!$omp end parallel

...

!$omp parallel

...

!$omp end parallel
Shared and Private Data

• Inside a parallel region, variables can either be *shared* or *private*

• All threads see the same copy of shared variables

• All threads can read or write shared variables

• Each thread has its own copy of private variables: these are invisible to other threads

• A private variable can only be read or written by its own thread
Parallel Loops

- In a parallel region, all threads execute the same code

- OpenMP has also directives which indicate that work should be divided up between threads, not replicated
  - this is called worksharing

- Since loops are the main source of parallelism in many applications, OpenMP has an extensive support for parallelising loops

- There are a number of options to control which loop iterations are executed by which threads

- It is up to programmer to ensure that the iterations of a parallel loop are independent

- Only loops where the iteration count can be computed before the execution of the loop begins can be parallelised in this way
Synchronisation

• The main synchronisation concepts used in OpenMP are:

• Barrier
  • all threads must arrive at a barrier before any thread can proceed past it
  • e.g. delimiting phases of computation

• Critical regions
  • a section of code which only one thread at a time can enter
  • e.g. modification of shared variables

• Atomic update
  • an update to a variable which can be performed only by one thread at a time
  • e.g. modification of shared variables (special case)
History

• Combined OpenMP C/C++/Fortran standart (2.5) released in May 2005
  • no new features, but extensive rewriting and clarification

• Version 3.0 released in May 2008
  • new features, including tasks, better support for loop parallelism and nested parallelism

• Version 3.1 released in June 2011
  • corrections and some minor new features
  • most current compilers support this

• Version 4.0 released in July 2013
  • accelerator offloading, thread affinity, more task support
  • now appearing in implementations

• Version 4.5 released in November 2015
  • corrections and a few new features
Compiling and Running OpenMP Programs

- OpenMP is built-in to most of the compilers you are likely to use.

- To compile OpenMP program you need to add a (compiler-specific) flag to your compile and link commands:
  - `-fopenmp` for gcc/gfortran
  - `-openmp` for Intel compilers

- The number of threads which will be used is determined at runtime by the `OMP_NUM_THREADS` environment variable.

  - Set this before you run the program.

  - E.g. `export OMP_NUM_THREADS=4`

- Run in the same way you would a sequential program:
  - Type the name of the executable.
Exercise

• “Hello World” program

• Aim: to compile and run a trivial OpenMP program

• Vary the number of threads using the OMP_NUM_THREADS environment variable

• Run the code several times. Is the output always the same?
Parallel Regions
Parallel Region Directive

• Code within a parallel region is executed by all threads

• Syntax:

C/C++: #pragma omp parallel
     {
       block
     }

Fortran: !$OMP PARALLEL
         block
         !$OMP END PARALLEL
fred();
#pragma omp parallel
{
    billy();
}
daisy();
Useful Functions

• Often useful to find out number of threads being used

Fortran:

USE OMP_LIB
INTEGER FUNCTION OMP_GET_NUM_THREADS()

C/C++:

#include <omp.h>
int omp_get_num_threads(void);

Note: returns 1 if called outside parallel region!
Useful Functions (cont.)

• Also useful to find out number of the executing thread

Fortran:

```fortran
USE OMP_LIB
INTEGER FUNCTION OMP_GET_THREAD_NUM()
```

C/C++:

```c
#include <omp.h>
int omp_get_thread_num(void);
```

Note: Takes value between 0 and `OMP_GET_NUM_THREADS() - 1`
Clauses

• Specify additional information in the parallel region directive through clauses:

C/C++: \#pragma omp parallel [clauses]

Fortran: !$OMP PARALLEL [clauses]

• Clauses are comma or space separated in Fortran, space separated in C/C++
Shared and Private Variables

• Inside a parallel region, variables can be either **shared** (all threads see same copy) or **private** (each thread has its own copy)

• **shared**, **private** and **default** are OpenMP clauses

C/C++:

```plaintext
shared(list)
private(list)
default(shared|none)
```

Fortran:

```plaintext
SHARED(list)
PRIVATE(list)
DEFAULT(SHARED|PRIVATE|NONE)
```
• On entry to a parallel region, private variables are uninitialised

• Variables declared inside the scope of the parallel region are automatically private

• After the parallel region ends, the original variable is unaffected by any changes to private copies

• Not specifying a DEFAULT clause is the same as specifying DEFAULT(SHARED)

  • Danger!

  • Always use DEFAULT(NONE)
Shared and Private (cont.)

- Example: each thread initializes its own column of a shared array

```
!$OMP PARALLEL DEFAULT (NONE), PRIVATE (I, MYID),  
!$OMP& SHARED(A,N)
  MYID = OMP_GET_THREAD_NUM() + 1
  DO I = 1, N
    A(I, MYID) = 1.0
  END DO
!$OMP END PARALLEL
```
Multi-line Directives

C/C++:

```c
#pragma omp parallel default(none) \ 
private(i,myid) shared(a,n)
```

Fortran: fixed source form

```fortran
!$OMP PARALLEL DEFAULT(NONE), PRIVATE(I,MYID),
!$OMP& SHARED(A,N)
```

Fortran: free source form

```fortran
!$OMP PARALLEL DEFAULT(NONE), PRIVATE(I,MYID), &
!$OMP  SHARED(A,N)
```
Initializing Private Variables

- Private variables are uninitialized at the start of the parallel region
- If we wish to initialize them, we use **FIRSTPRIVATE** clause:

  C/C++: `firstprivate(list)`

  Fortran: **FIRSTPRIVATE(list)**
b = 23.0;
.
.
.
.
#pragma omp parallel firstprivate(b),
private(i, myid)
{
    myid = omp_get_thread_num();
    for (i=0; i<n; i++) {
        b += c[myid][i];
    }
    c[myid][n] = b;
}
Reductions

• A *reduction* produces a single value from associative operations such as addition, multiplication, max, min, and, or

• Would like each thread to reduce into a private copy, then reduce all these to give final result

• Use **REDUCTION** clause:

  C/C++: `reduction(op: list)`

  Fortran: `REDUCTION(op: list)`

• Can have reduction arrays in Fortran, but not in C/C++
Reductions (cont.)

\[
B = 10
\]

\[
!\text{OMP PARALLEL REDUCTION (+:B)},
!\text{OMP& PRIVATE(I, MYID)}
\]

\[
\text{MYID} = \text{OMP\_GET\_THREAD\_NUM()} + 1
\]

\[
\text{DO I = 1, N}
\]

\[
B = B + C[I][MYID]
\]

\[
\text{END DO}
\]

\[
!\text{OMP END PARALLEL}
\]

\[
A = B
\]
Worksharing
Worksharing Directives

- Directives which appear inside a parallel region and indicate how work should be shared out between threads are:
  - Parallel DO/for loops
  - Single directive
  - Master directive
Parallel DO/for Loops

• Loops are the most common source of parallelism in most codes. Therefore, parallel loop directives are very important!

• A parallel DO/for loop divides up the iterations of the loop between threads

• The loop directive appears inside a parallel region and indicates that the work should be shared out between threads, instead of replication

• There is a synchronisation point at the end of the loop: all threads must finish their iterations before any thread can proceed
Parallel DO/for Loops (cont.)

Syntax:

Fortran:  

\$OMP DO [clauses] 
  DO loop 
  \$OMP END DO

C/C++:  

#pragma omp for [clauses] 
  for loop
Parallel Loops (Example)

```c
!$OMP PARALLEL
$OMP DO
   DO i=1,n
      b(i)=(a(i)-a(i-1))*0.5
   END DO
$OMP END DO
$OMP END PARALLEL

#pragma omp parallel
{
   #pragma omp for
   for (int i=1, i<=n, i++) {
      b(i)=(a(i)-a(i-1))*0.5;
   }
}
```
This construct is common that there is shorthand form which combines parallel region and DO/for loops

C/C++: #pragma omp parallel for [clauses]
for loop

Fortran: !$OMP PARALLEL DO [clauses]
do loop
!$OMP END PARALLEL DO
• DO/for directive can take PRIVATE, FIRSTPRIVATE and REDUCTION clauses which refer to the scope of the loop

• Note that the parallel loop variable is PRIVATE by default
  • loop indices are private by default in Fortran, but not in C

• PARALLEL DO/for directive can take all clauses available for PARALLEL directive

• PARALLEL DO/for is not the same as DO/for or the same as PARALLEL
• With no additional clauses, the DO/for directive will partition the iterations as equally as possible between the threads

• However this is implementation dependent, and there is still some ambiguity

e.g. 7 iterations, 3 threads. Could partition as 3+3+1 or 3+2+2
The SCHEDULE clause gives a variety of options for specifying which loop iteration are executed by which thread.

Syntax:

C/C++: `schedule(kind[, chunksize])`

Fortran: `SCHEDULE(kind[, chunksize])`

where kind is one of

```
STATIC, DYNAMIC, GUIDED, AUTO or RUNTIME
```

and chunksize is an integer expression with positive value.

• e.g. `!$OMP DO SCHEDULE(DYNAMIC, 4)`
• With no *chunksize* specified, the iteration space is divided into (approximately) equal chunks, and one chunk is assigned to each thread in order (**block schedule**)

• If *chunksize* is specified, the iteration space is divided into chunks, each of *chunksize* iterations, and the chunks are assigned cyclically to each thread in order (**block cyclic** schedule)
STATIC Schedule

SCHEDULE (STATIC)

SCHEDULE (STATIC, 4)
DYNAMIC Schedule

- DYNAMIC schedule divides the iteration space up into chunks of size \textit{chunksize}, and assigns them to threads on a first-come-first-served basis.

- i.e. as a thread finish a chunk, it is assigned the next chunk in the list.

- When no \textit{chunksize} is specified, it defaults to 1.
GUIDED Schedule

• GUIDED schedule is similar to DYNAMIC, but the chunk starts off large and gets smaller exponentially

• The size of the next chunk is proportional to the number of remaining iteration divided by the number of threads

• The \textit{chunksize} specifies the minimum size of the chunks

• When no \textit{chunksize} is specified, it defaults to 1
DYNAMIC and GUIDED Schedules

\[ \text{SCHEDULE(DYNAMIC, 3)} \]

\[ \text{SCHEDULE(GUIDED, 3)} \]
Choosing a Schedule

- STATIC best for load balanced loops - least overhead
- STATIC, n good for loops with mild or smooth load imbalance, but can induce overheads
- DYNAMIC useful if iterations have widely varying loads, but ruins data locality
- GUIDED often less expensive than DYNAMIC, but beware of loops where the first iterations are the most expensive
- AUTO may be useful if the loop is executed many times over
SINGLE Directive

• Indicates that a block of code is to be executed by a **single thread only**

• The first thread to reach the SINGLE directive will execute the block

• There is a **synchronisation** point at the end of the block: all other threads wait until block has been executed
SINGLE Directive (cont.)

Syntax:

Fortran

\[
!$OMP \text{ SINGLE} \ [\text{clauses}] \\
\text{block} \\
!$OMP \text{ END SINGLE}
\]

C/C++:

\[
\#\text{pragma omp single} \ [\text{clauses}] \\
\text{structured block}
\]
#pragma omp parallel
{
    setup(x);
    #pragma omp single
    {
        input(y);
    }
    work(x,y);
}
• SINGLE directive can take PRIVATE and FIRSTPRIVATE clauses

• Directive must contain a structured block: can not branch into or out of it
• Indicates that a block of code should be executed by the **master thread** (thread 0) **only**

• There is **no synchronisation** at the end of the block: other threads skip the block and continue executing
MASTER Directive (cont.)

Syntax:

Fortran

\[ !$OMP MASTER \]
block
\[ !$OMP END MASTER \]

C/C++:

\[ \#pragma omp master \]
structured block
Synchronisation
Why is It Required?

- Need to synchronise actions on shared variables
- Need to ensure correct ordering of reads and writes
- Need to protect updates to shared variables (not atomic by default)
BARRIER Directive

• No thread can proceed reached a barrier until all the other threads have arrived

• Note that there is an implicit barrier at the end of DO/for, SECTIONS and SINGLE directives

• Syntax:

  C/C++: #pragma omp barrier

  Fortran: !$OMP BARRIER

• Either all threads or none must encounter the barrier: otherwise DEADLOCK!
Example:

```c
!$OMP PARALLEL PRIVATE(I,MYID,NEIGHB)
  myid = omp_get_thread_num()
  neighb = myid - 1
  if (myid.eq.0) neighb = omp_get_num_threads()-1
  ...
  a(myid) = a(myid)*3.5
  !$OMP BARRIER
  b(myid) = a(neighb) + c
  ...
!$OMP END PARALLEL
```

- Barrier required to force synchronisation on `a`
Critical Sections

- A critical section is a block of code which can be executed by only one thread at a time.

- Can be used to protect updates to shared variables.

- The CRITICAL directive allows critical sections to be named.

- If one thread is in a critical section with a given name, no other thread may be in a critical section with the same name (though they can be in critical sections with other names).
Critical Directive

Fortran

```
!$OMP CRITICAL [(name)]
block
!$OMP END CRITICAL [(name)]
```

C/C++:

```
#pragma omp critical [(name)]
structured block
```

- In Fortran, the names on the directive pair must match.
- If the name is omitted, a null name is assumed (all unnamed critical sections effectively have the same null name).
QUESTIONS
or
COMMENTS!